Simultaneous Localization and Mapping with Atlas

Submitted 05/06/2021

—

Ryan Carnemolla (RBE), Anirban Mukherjee (RBE/ECE),
Amrit Parmanand (CS), and Emily Staknis (RBE)

Advised by Professor Michael Gennert

Abstract

Simultaneous Localization and Mapping (SLAM) is a powerful method for autonomously exploring new environments, which has never been applied to humanoid robots. This project implemented SLAM on the Atlas humanoid robot, allowing it to interact with and explore novel environments. The Atlas robot was programmed to create 3 dimensional maps of environments and localize itself within them. Due to constraints in time and safety the project was conducted mostly in simulation. Both stereo cameras and LiDAR were used in experiments to identify the robot's surroundings and avoid obstacles. These sensors paired with the robot's locomotion allowed the Atlas robot to search and navigate unknown areas.

Acknowledgements

To Professor Michael Gennert

Thank you for giving our team the opportunity to work on a project so interesting and with a robot as groundbreaking as Atlas.

To Ashay Aswale

Ashay, your continued support and mentorship in the laboratory has been a great help to our team and for that we are deeply appreciative.

To Katherine Crighton

Because of your tireless efforts to get us into the laboratory we are able to work with equipment we've only ever dreamed of.

Executive Summary

As technology evolves and improves, so does the feasibility and practicality of robotic solutions for problems facing the world today. Humanoid robotics research has opened the possibility of simulating a wide variety of complex human capabilities through robotics. The ability to adapt to unique circumstances is one of humanity's strongest characteristics, but it is one that is difficult to implement through robotics. When in a new environment, humans are able to gather information and understand their surroundings quickly, a capability that is essential for robots to master if they are to be used in place of humans in, for instance, unknown and dangerous environments like crisis areas and disaster cites. To make humanoids interact robustly with new and altered environments, they must be able to simultaneously create a high quality 3-dimensional map of the nearby environment and localize themselves in the map.

Background:

To create dynamic 3D mapping on the Atlas robot, the Simultaneous Localization and Mapping (SLAM) method was used. Previous research with the Atlas robot and other humanoids has focused on topics such as path planning, locomotion, and obstacle avoidance, all of which are complex tasks. With a base of similar research done in the WPI Humanoid Robotics Lab (WHRL), this project was able to focus on high-level programming of SLAM on the Atlas robot. Other open-source code was used to complete different elements of the project, like Robotic Operating System (ROS) and various libraries, including Octomap for grid based 3D mapping, and Viso2 for visual odometry. The Atlas robot itself is a complex 30 degree of freedom (DOF) robot with a variety of sensors critical to this project, like the stereo cameras and LiDAR.

Methodology:

To establish this ability, a mapping algorithm was created, which uses output from Atlas' LiDAR as the robot moves to create a map. To do so, various external libraries were used, which performed necessary tasks, ranging from map generation to visualization. After using these to create a foundation for new code, the various tasks and steps of SLAM were created in ROS. These localization and mapping techniques work in real time and can account for uncertainties in the height and angle of the robot's head. An exploration algorithm was written to direct the robot to autonomously explore and map its environment using the TOUGH library movement and path planning capabilities. This algorithm allowed us to test the speed and performance of SLAM in simulation and the capabilities of the different sensors Atlas is equipped with.

Results and Discussion:

In doing so, Atlas is able to fully explore and generate maps of enclosed environments. The final map is robust and accurate, with loop closure and few artifacts. The two visual sensors on the robot's multisense head both had their benefits and drawbacks, but it was found

that the LiDAR was much more suited to the construction of maps required for SLAM. By using it and implementing custom ROS code, complete maps of fully enclosed areas could be generated. We were also able to follow a process for remote development of humanoid robotics research through trials of solving different bugs and issues. Through the duration of the project, many issues were encountered, including difficulties setting up various different libraries and controllers that needed to work in tandem and getting them to work, as well as many others, all of which will be discussed along with their solutions. Nevertheless, our project ended with the option to continue research in multiple different directions.

Conclusion:

Through conducting our research in this project, we were able to implement novel research and to understand the process necessary to work with humanoid robots. The many new directions for future work from our study make it possible to continue researching SLAM on Atlas to develop a stronger response to novel environments. This would be a strong capability that motivated this project, so pursuing it further could lead to even better performance in the future.

Authorship

Ryan Carnemolla:

Wrote the Executive Summary, most of the introduction, and aided in the writing of the Abstract. Wrote multiple sections of the Background. Wrote multiple sections of the Methodology. Contributed to the Results and Discussion. Aided in the editing and formatting of the paper as a whole.

Anirban Mukherjee:

Wrote part of the background, aided in the writing of the abstract, organized general formatting, wrote part of the methodology, and wrote the conclusion.

Amrit Parmanand:

Wrote part of the troubleshooting section, aided in the formatting and editing of the paper.

Emily Staknis:

Wrote part of the literature review, aided in writing the abstract, the mapping section of the methodology, the results, and the successes and shortcomings section of the discussion and analysis. Also added appendices and worked on general formatting and editing of the paper.

From left to right: Anirban Mukherjee, Emily Staknis, the ATLAS Robot, Ryan Carnemolla, Amrit Parmanand

Table of Contents

Introduction

Humanoid robots have a huge potential to help in the automation of tasks that are dangerous and otherwise unreasonable for humans. Because of their similarities to us, they can be, hypothetically, applied to any risky situation that humans find themselves in. However, the field of humanoid robotics research has a long way to come before robots are able to seamlessly perform the extremely versatile variety of jobs people do. Humanoids are expensive and complex, require a large amount of processing power, and need to react to a broad amount of situations. This is a unique problem for robots, which are optimally used in repetitive and well-defined circumstances. While humans are able to respond quickly in novel situations, robots require at least a degree of preparation to respond to different stimuli. This requires robust programming, particularly for loosely defined situations.

For instance, disaster sites, like areas hit by earthquakes, flooding, or explosions, are often unknown environments that require humans to put themselves in danger to respond. Robots can be applied to such situations without risking further human lives. However, there are a large amount of uncertainties in disaster sites that create difficulties for robots. Among them is the unknown nature of the area. A human can easily look around and understand where they are within an area and robots that need to respond to broad issues without prior knowledge must have a similar level of adaptive capability. To do so, the robot must be able to generate a 3D map of its surroundings and localize itself within it. This requires the robot to be programmed with such capabilities.

Purpose

Simultaneous Localization and Mapping (SLAM) is the process robots use to understand and position themselves within their surroundings. The focus of this project was developing SLAM on the Boston Dynamics Atlas humanoid robot, to allow it to interact with novel environments. Atlas is among the most advanced in the field and the Warner model has been used by WPI for humanoid robotics research since the Defense Advanced Research Projects Agency (DARPA) Robotics Challenge in 2013. Through the use of open source robotics middleware Robot Operating System (ROS) and the TOUGH libraries, developed between WPI and the Institute for Human and Machine Cognition (IHMC) for smooth control of the robot, Atlas was programmed to dynamically generate maps of novel environments and locate itself within them. This allowed it to explore and navigate unknown areas.

Implementing SLAM on a humanoid robot is a novel concept that has not seen a lot of development. Nonetheless, SLAM is a powerful tool for interacting with unknown environments. Because humanoid robots have the potential to perform the same tasks as humans do, their responses need to be dynamic and adaptive, like people. While creating complex human-like behavior was beyond the scope of this project, its purpose was to develop the ability for humanoid robots to react to unique, unknown situations through SLAM. The project sought to create basic slam functionality in simple novel environments and to explore the benefits and drawbacks of various methods of implementing SLAM, including the use of different sensors and approaches to localization.

Goals and Objectives

The specific goal for this project was to develop a SLAM algorithm that the Atlas robot could use to map its surroundings while understanding its position within them. In order to keep the project on schedule, the team established a list of realistic objectives to use as milestones of the progress made throughout the four terms of work. These were initially brainstormed, and were adapted as time went on, and as the team became more familiar with the steps needed to be taken in order to make these possible.

The initial objective was to organise and set up the requirements for the project. Over the course of the first two months, there was one main objective: creating a foundation for the rest of the project. This included setting up development environments on personal computers and on the computers in the WPI Humanoid Robotics Lab (WHRL), researching previous work with humanoid robots inside and outside of WPI to refine the project scale, and developing an understanding of the existing code infrastructure available for use during the project.

Upon completing this objective, the project focus shifted to creating the SLAM algorithm itself. Initially, this required implementing various libraries to complete different tasks necessary to SLAM. Early in this process, complex libraries were used in an attempt to implement SLAM quickly, allowing for the project to focus on optimizing the result. However, difficulties using external libraries like this made it clear that implementing an original application of SLAM was more practical and academically valuable.

As a consequence, the objective shifted to generating mapping from scratch. The objectives used to do this included implementing visual odometry and map transformations, generating both 2-D and 3-D graphs, frontier based exploration, and code coordination with the TOUGH library. These were achieved with various levels of success, leading to the completion of a SLAM algorithm which allowed Atlas to map and localize within novel environments.

Our final objective was to test the success of the algorithm and the benefits and drawbacks of different methods attempted, including the use of different visual sensors. This report will detail the background that informed this project, the repeatable process used to complete it, the results observed, and the importance of said results.

Literature Review

During the preparation for this study, literature and previous work pertaining to several topics were reviewed to contextualize it. These articles and studies discussed topics including, but was not limited to, the Atlas robot's mechanics and sensors, previous work with Atlas, humanoid robotic locomotion, and methods for localization and mapping. In reviewing these materials, a useful base of knowledge was created, which also works as a background of information for understanding this project.

The Atlas Robot

The Atlas robot was created by Boston Dynamics and is a 30 degree of freedom humanoid robot [14]. It was at the center of the DARPA Robotics Challenge in which teams had to program the robot to climb into and out of a utility vehicle, walk on difficult terrain, and perform some manipulation task [14]. The Werner Atlas robot used for this study was gifted to WPI by Boston Dynamics for this challenge and for continued humanoid robotics research. Though the tasks of the DARPA challenge proved challenging to complete, Boston Dynamics has continued their research and made incredible progress, recently programming Atlas to perform a backflip. For many other researchers, getting Atlas to walk proves a difficult enough challenge.

Transportable OpenSource API & UI for Generic Humanoids

In the DARPA robotics challenge, many teams wrote a set of controllers for the robot. The controllers allowed the teams to command the robot to complete specific tasks. The controllers were almost always specific to the challenges faced at the DARPA challenge, such as climbing stairs. The Institute for Human and Machine Cognition (IHMC), however, wrote more generalized controllers that lend themselves well to almost any task the robot may need to complete. The Transportable OpenSource API & UI for Generic Humanoids (TOUGH) was developed to use IHMC controllers and to further facilitate control of humanoid robots.

TOUGH provides users with some basic controllers that allow the robot to remain stationary, balance, and move in a very basic way (i.e. walking in a straight line, lifting an arm). TOUGH's work in the background of robot operation can be compared to the many sensors and systems in the automotive industry. A car now comes equipped with traction control, lane keep assist, and even more safety settings that protect the driver and passengers. Similarly, TOUGH comes equipped with a framework that prevents issues like joints locking up or tripping on relatively even terrain. TOUGH also provides users with a Graphical User Interface with which they can edit parameters of the robot and observe it in simulation.

TOUGH also provides users with very basic navigation using A*. The A* treats any object which is above the flat terrain as an obstacle, regardless of if it is 1 inch or 1 foot off the ground. TOUGH also has a perception library through which we can access Atlas' LiDAR and Stereo Camera data. More information on Atlas' sensors is below. Though we won't be using it, the library also has gripper control algorithms.

For all these reasons, TOUGH provides our team with an excellent toolkit for development. We will therefore be using it in tandem with our own code for SLAM in order to reach our end results.

Locomotion

A variety of papers have focused on improving and expanding Atlas' ability to traverse both flat and uneven environments. Early research on humanoid robotics defined a "Linear Inverted Pendulum Model" for biped robots, which described a linear model for a biped robot's gait when the robot's center of mass remained at a constant height [20]. The linear model is often used even if the robot's center of mass is not at a perfectly constant height. Thus, it is useful for flat or nearly flat terrain, but does not necessarily apply to terrain which is not flat [19]. For large step-ups, it is necessary to follow a non-linear model because the change in height of the center of mass is not negligible [19].

The Atlas robot at WPI can walk using a Q-learning algorithm trained in a previous MQP [15]. The group trained Atlas in simulation to walk on flat ground by "grading" the robot on, among other things, if it fell and how long it took to traverse the environment [15]. The WPI Atlas robot can consistently traverse flat terrain by shifting its weight completely to one leg and waiting until the other foot is completely on the ground in a new position before shifting the weight back.

Some research has centered around more complex topics. One paper succeeded in having the robot walk on a series of partial footholds [18]. Many of these footholds were line contacts [18]. They explored the potential foothold ahead of time with the robot's foot, and from this exploration determined the optimal center of pressure needed to maintain the robot standing [18]. They also explored upper body balance techniques to better control the robot's center of mass and prevent the robot from falling without needing to put a second foot down [18].

Atlas' Sensors

Atlas' head is a Carnegie Mellon Multi-sense head. The Multi-sense head has a Hokuyo UTM-30LX-EW planar Light Detection and Ranging (LiDAR) sensor that can capture 40 scan lines of the environment per second with a maximum range of 30m (Kuindersma et al). LiDARs fire lasers at objects and read reflected light to deduce three-dimensional information in the form of a point cloud. LiDAR lacks the spatial resolution offered by cameras and is slower than that of cameras because it requires greater amounts of pre-processing. This is often due to the amount of noise in LiDAR data, but also from the mechanical limitations of most LiDAR devices. The LiDAR module on the MultiSense head contains an array of infrared sensors that revolve around an axis at given intervals. By spinning these infrared arrays the module “scans” the robot’s immediate environment. Consequently, the rate of data collection is limited by the speed at which this sensor rotates about its axis.

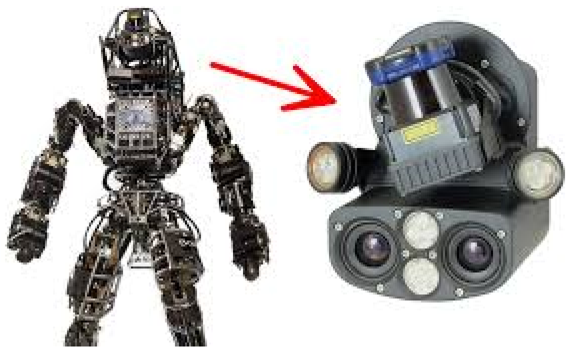

Figure 1: Atlas MultiSense head [16]

The head also has two stereo cameras which can provide color data and point cloud data [16]. Stereo cameras use images from both cameras as reference points for perceiving depth. Mathematical post-processing can ultimately deduce distance of an object from this method. The ability of cameras to perceive color has allowed machine learning algorithms to perform object detection. The use of this technology is how Atlas was able to open doors and turn valves in the DARPA robotics challenge.

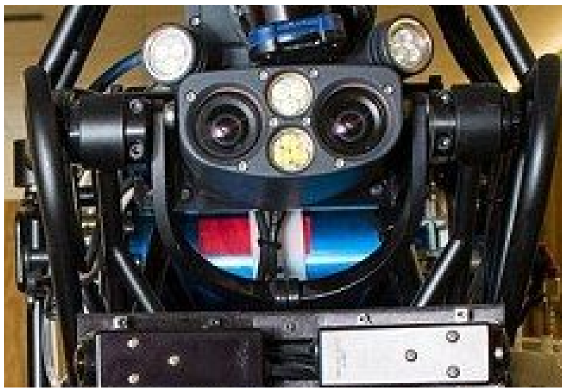

Figure 2: Atlas Stereo Cameras [16]

The head has its own ROS drivers and topics [21]. The topics publish raw and processed output from the sensors, including images from the stereo camera, depth images (not published when using gazebo), camera info, point clouds from the LiDAR and from the stereo cameras, and calibration information.

The manipulators at the end of Atlas' arms are exchangeable. Some manipulators are similar to hands and have haptic sensors in the palms and fingers. Others are less complicated and have cameras in the center of the hand to provide a better view of the object to be manipulated. There are also cameras on each of Atlas' ankles pointing forward to provide a better view of the upcoming terrain.

Atlas also has an IMU placed at the base of where Atlas' tailbone would be. Each of Atlas' joints have sensors that read the angle of the joints. Atlas' feet have pressure sensors. These sensors can sense pressure from pushing on the bottom or the top of Atlas' feet [17].

Point Clouds and the Point Cloud Library

Point clouds are a message type that will be prolific in the assembly of maps for this project as they've been the standard for SLAM algorithms, especially when using ROS. The structure of a point cloud is the following: a binary cloud of discretized points in free space. The way these point clouds are discretized depends on the way the ROS data type is structured. There are two point cloud data types in ROS, and they are pointCloud and pointCloud2. Both are from the sensor_msgs library.

PointCloud messages are composed of a header, an array of 3d points, and an array of channels. Details can be seen below:

```
# This message holds a collection of 3d points, plus optional additional
# information about each point.

# Time of sensor data acquisition, coordinate frame ID.
Header header

# Array of 3d points. Each Point32 should be interpreted as a 3d point
# in the frame given in the header.
geometry_msgs/Point32[] points

# Each channel should have the same number of elements as points array,
# and the data in each channel should correspond 1:1 with each point.
# Channel names in common practice are listed in ChannelFloat32.msg.
ChannelFloat32[] channels
```

The pointCloud message type is useful for interpreting standard point clouds, but they are not capable of providing information such as depth, intensity, and color. This is largely why pointCloud2 has been adopted by the multisense interface instead. Both the LiDAR and stereo cameras return pointCloud2 topics, which we use in this project to assemble a 3D Occupancy grid. The pointCloud2 structure can be seen below:

```
# This message holds a collection of N-dimensional points, which may
# contain additional information such as normals, intensity, etc. The
# point data is stored as a binary blob, its layout described by the
# contents of the "fields" array.

# The point cloud data may be organized 2d (image-like) or 1d
# (unordered). Point clouds organized as 2d images may be produced by
# camera depth sensors such as stereo or time-of-flight.

# Time of sensor data acquisition, and the coordinate frame ID (for 3d
# points).
Header header

# 2D structure of the point cloud. If the cloud is unordered, height is
# 1 and width is the length of the point cloud.
uint32 height
uint32 width

# Describes the channels and their layout in the binary data blob.
PointField[] fields

bool    is_bigendian # Is this data bigendian?
uint32  point_step   # Length of a point in bytes
uint32  row_step     # Length of a row in bytes
uint8[] data         # Actual point data, size is (row_step*height)

bool is_dense        # True if there are no invalid points
```

It's clear that the pointCloud2 message provides more flexibility in its ability to characterize discrete points. In the multisense interface, the pointCloud2 data fields for the stereo cameras are denoted as [x,y,z,RGB]. These fields are stated in the PointField[ ] array called "fields". To ensure that the data frames are being passed correctly through the ROS environment, the message also contains a height and width so a subscriber can check whether or not the message is complete.

The LiDAR messages are brought up through the tough perception nodes, specifically the ones in the launch file “field_laser_assembler.launch”. This file is inside the tough perception package. The LiDAR iterates the point clouds in frames, and then publishes to the topic field_assemledCloud2. With a real time factor of around 0.2 in Gazebo, it takes about 30-40 seconds for these pointClouds to appear in Rviz. The perception nodes will also automatically stop the LiDAR from publishing for a certain duration of time in order to protect the system from memory overflow. This “pause” is also controlled through a rostopic that contains a Boolean which can enable and disable this feature.

Mapping Methods

Mapping environments using Atlas or humanoid robots in general is not a common occurrence. Most environments that Atlas traverses are pre-mapped by a different robot. In part, this allows for extensive testing in simulation and real-life testing in a comparable environment.

In Kuindersma et al, though, Atlas' LiDAR is used for initial mapping. They begin by having the robot stand still for 30 seconds to collect point cloud data of the environment. The cloud is then converted into an occupancy grid. The robot uses the map to move around. The article states that "if the robot were to approach the map boundary, on-line construction of a new map could easily be performed during operation" (Kuindersma et al).

This differs from SLAM since they do not constantly update the map, but rather update it only when missing information is required [6]. They also have the robot stand still for 30 seconds at the start to build the initial map [6]. This causes the movement of the LiDAR to be negligible and thus avoids many of the problems with mapping using humanoids.

Some other researchers got around this issue by placing the sensors elsewhere on the robot or designing the robot's gait such that the sensor always stays at the same angle and height.

Another issue addressed by [13] is representing a 3D environment in 2D or 2.5D, a height map. They describe an example in which the LiDAR views a table on a floor and averages the heights in the height map. They continued to use 2.5D, but used standard deviation to find clusters in the point cloud and took the height to be the average of the highest cluster [13]. Other researchers created 3D occupancy maps and worked with these instead [4].

Localization Methods

Localization for wheeled robots can be completed in 2 dimensions due to the wheeled robot's inability to climb, step over, and crouch under obstacles. Multiple methods have worked to improve maps for humanoid robots, from 2.5D maps to placing the LiDARs in the robot's feet [4]. Humanoid robots trying to perform complex maneuvers, however, require 3D maps of their environment.

Localization can be complicated by the humanoid tilting the sensors as it moves. The angle of the sensors can be estimated using joint angles and forward kinematics, but in order to be accurate the angle between the robot's feet and the ground must be known. As a result, some studies ignore the forward kinematics in part or completely and instead use IMUs, even very noisy ones, to estimate the angle of the sensor [7]. This is especially relevant while the robot is walking.

Particle filters have been used in conjunction with Kalman filters on Atlas in past research [6], and have been used on other humanoid robots on many occasions for localization [7,8,9]. Particle filters are an effective method of localizing a robot that has noisy movement and measurements. It begins by randomly distributing simulated robots over a map and then moving the robots in the same way as the real robot moves. The particle filter becomes survival of the fittest. The robots are redistributed on the map, with higher probability particles gaining robots and lower probability particles losing robots. The probabilities are found by comparing the real and the simulated robot sensor readings.

The Rao-Blackwellized Particle Filter (RBPF) was used to reduce the number of particles and computational time. RBPF uses a Kalman filter localization with the particle filter [3]. The Kalman Filter estimates an area in which the robot is likely to be based on the past movements of the robot. KBPF then only scatters particles in the area that the Kalman filter returns [3].

Some researchers used a global particle filter to first localize the robot and then optimized and reduced the number of particles once the robot had been found [4]. Often, the particles are simulated as wheeled robots instead of humanoid robots to simplify the localization problem [8]. Reducing the accuracy of the gaussian curve also significantly improved computation time for real-time localization of the robot [8].

Using sensor fusion is another way to improve the accuracy of localization. Combining the information from LiDARs, vision, and pressure and magnetic sensors when available can improve the localization of the robot. One paper, however, described a method using only a neural network, IMU data, and actuator load data as input that had high accuracy for small lightweight humanoids [11]. Their method performed significantly better than pure odometry, but would require additional research to determine if it could be applied to larger scale humanoids.

Pathfinding Algorithms

Once its environment is mapped and its location known, Atlas, and any other robot performing SLAM, must find the optimal path through the environment to its destination. To do this a pathfinding algorithm must be used.

A* is the most commonly used path finding algorithm due to its flexibility, efficiency, and ability to find the optimal path. In the simplest terms, A* is a mix between Djikstra’s algorithm and greedy best-first search.

Dijkstra's algorithm is usually used to find the shortest path from an initial point to all other points in a graph. It does this by examining the path costs of neighboring cells and assigning the lowest found path cost as the shortest path. The disadvantage of Dijkstra's algorithm is that it must explore a significant portion of the map to ensure it has the shortest path.

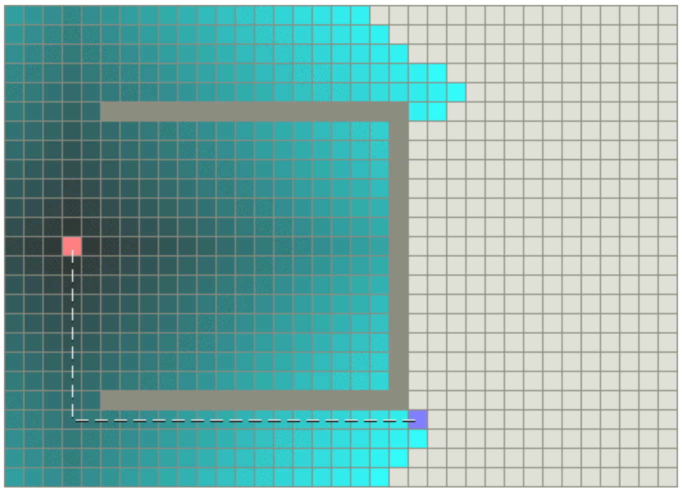

Figure 3: Dijkstra's Algorithm With Obstacles [12]

Greedy-best first search uses a heuristic to choose the next cell to explore. In certain environments, this allows for the shortest possible route to the destination with the least amount of effort, but introducing obstacles can cause the path to deviate from the shortest path [12].

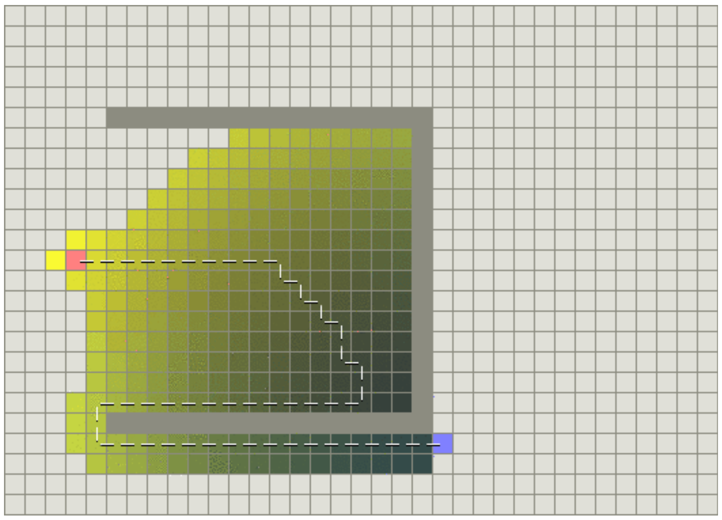

Figure 4: Greedy Best First Search With Obstacles [12]

A* makes use of a heuristic, but also takes into account the path length from the start node to the current node, like Dijkstra's algorithm. This results in an optimal algorithm that can be extremely useful in static environments [12].

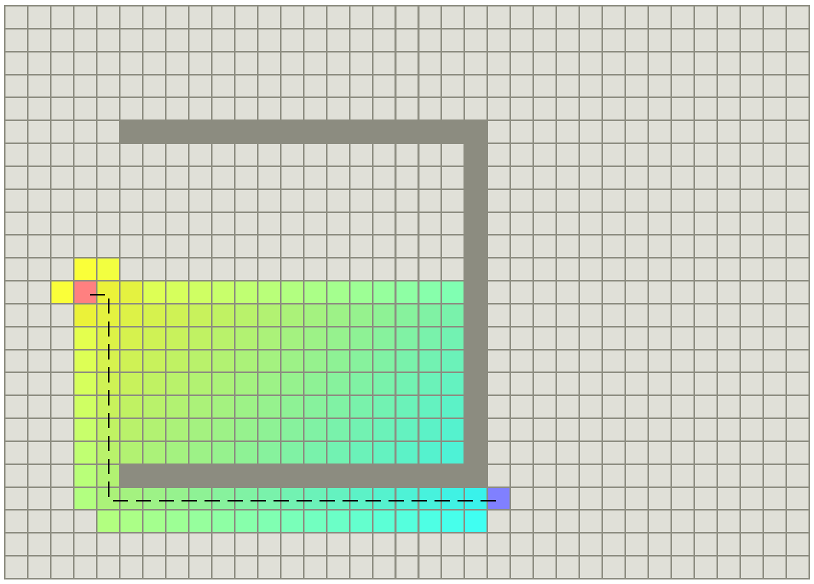

Figure 5: A* Algorithm With Obstacles [12]

A* has been used extensively in previous research for path-finding. However, in environments that are not static, other algorithms can be more useful. In *A Roadmap-based Planner for Fast Collision-free Motion in Changing Environments,* the D* algorithm was used, which updates the map information upon observing new circumstances, like moving obstacles. This allows for the path to be planned around newly observed information. Other previous projects have used Rapidly exploring Random Trees (RRT) as a search algorithm. This is particularly useful for pathfinding in open 3D spaces, whose random search helps mobile robots avoid getting stuck in unknown environments. FInding an appropriate search algorithm is critical to allowing Atlas to navigate environments once it knows the start and end location.

Methodology

This section of the report will highlight the procedure in which our project was completed, as well as the methods used to determine our results. First, the External Libraries & Software used for this project will be covered with regards to their utility. Additionally we'll cover the different ways we assembled our maps, and how we handled pathfinding with the robot.

External Libraries & Software

This project made use of several external libraries and packages. The most noteworthy are those we used for controlling the robot, computer vision, working with point clouds, and mapping. The implementations vary based on whether or not point clouds were assembled by LiDAR or Stereo Imaging.

TOUGH

As mentioned in the literature review section above, TOUGH is an API and GUI for interaction with humanoid robots. More prominent in this project however, is TOUGH's ability to bridge the low level Java delegates on Atlas with the architecture of ROS. As a result, TOUGH is essential as a controller and platform for simulating the Atlas robot. TOUGH also includes a footstep planner which organizes the robot's footsteps to reach a 2D navigation goal, or a goal location.

TOUGH's setup involves using a .yaml file for installation, from the repository. The file sets up local repositories on the host computer for both the IHMC controllers and the TOUGH GUI and API. By default, the installed version of the TOUGH repository will be cloned into the user's catkin workspace in the 'src' folder.

The default branch of the TOUGH repository runs the DARPA Robotics Challenge simulation. Our team used Gazebo for our implementation of SLAM, and so it was advisable to use the latest working branch of the Tough_Gazebo package. Launching the simulation thereafter involves editing the 'atlas_default_world.launch' file in the repository locally so the robot can be brought up in any environment.

The stereo cameras start publishing as soon as the simulation is initialized. In order to start the LiDAR however the spindle speed must be set manually for the LiDAR array, and then the field_laser_assembler. The laser scan takes 30 seconds in simulation time to publish data. Eventually the LiDAR publisher will automatically pause for a duration of time to prevent overflowing the CPU of the host machine.

The footstep planner can only be started after the LiDAR has been enabled. When running the tough_gui it's possible to then send nav goals to the robot and begin path planning using the octomap's 2D component.

The octomap functionality of TOUGH is also present within a launch file in the tough_perception package. This octomap can be edited locally on the host machine to control the operations for the octomap server.

As of the submission of the project, TOUGH is scheduled to be deprecated as a result of ROS kinetic no longer being supported, likely by September 2021. As a result, this implementation of SLAM will not be possible through the use of this software from that point onwards.

ROS

ROS provided the overall structure for the code generated in this project. It allowed us to have many separate tasks running concurrently and for us to pass data between tasks effectively. Both the TOUGH library and previous work the team had done with SLAM operated with ROS as well, which allowed us to efficiently create our code and coordinate it with the existing framework. The version of ROS being used is kinetic, because it's the only version that's compatible with TOUGH, meaning the cde was created in and must be run in a Ubuntu 16.04 environment. ROS will be dictating the way the code is structured, as it is necessary to make use of nodes and publisher/subscriber relationships.

Because LiDAR data is much more accurate in 3D mapping, the LiDAR based structure for SLAM:

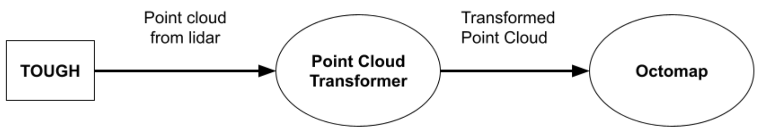

Figure 6: Map Generation Node Graph

ROS installation is well documented on the ROS wiki. Steps include setting up sources list, getting installation keys, installing through sudo apt, setting up the environment, and installing all dependencies for packages.

The main rostopics being used from TOUGH are from the stereo cameras and the LiDAR. This data is fed from those topics to our mapping nodes. Using the tough perception packages for default bringup renders point clouds for both sensors. These topics are recorded as a PointCloud2 datatype. This functionality, along with that of the footstep planner and the gazebo simulation are the primary tools for working on our SLAM implementation.

Viso2

The Viso library is a package used for performing visual odometry on stereo images. It was used in an attempt to calculate an estimated pose for the robot within the map it created. Viso also provides us with odometry on how far the robot has moved and in what direction it moved in. Upon running Viso and mapping its subscribed topics to listen to the output of the robots cameras, the library returns a set of cartesian and quaternion values that indicate the pose, which would allow for a homogeneous transformation matrix to be created, which would represent the change in position form the origin, in this case being the robots starting point.

We hoped to use visual odometry to verify the robot's position with respect to its environment, and to transform incoming point cloud topics to be added to a global XYZ coordinate frame. We hoped that this would provide us with accurate data on the robot's location.

Installing Viso2 is as simple as adding the package to your catkin workspace's source folder and building. Afterwards it requires remapping of the subscriber rostopic to whatever topic the raw images from the camera are publishing in the viso2 roslaunch file.

Octomap Library

The octomap library is included in the tough_perception commons, so our team needed only to install the RVIZ plugins for visualizing octomap's 3D occupancy grid. Having this capability is imperative for our team because we verified the completeness and accuracy of our created maps through RVIZ. The octomap_server library is an extremely useful open source library that utilizes the 3D map format of octomap. The server acts as a ROS node that listens to point clouds being published and returns various forms of maps, including both the 3-D gridded octomap, converted from the point cloud, and a down-projected 2-D map that consists of every occupied grid cell of the 3-D map being marked as an obstacle on the 2-D map.

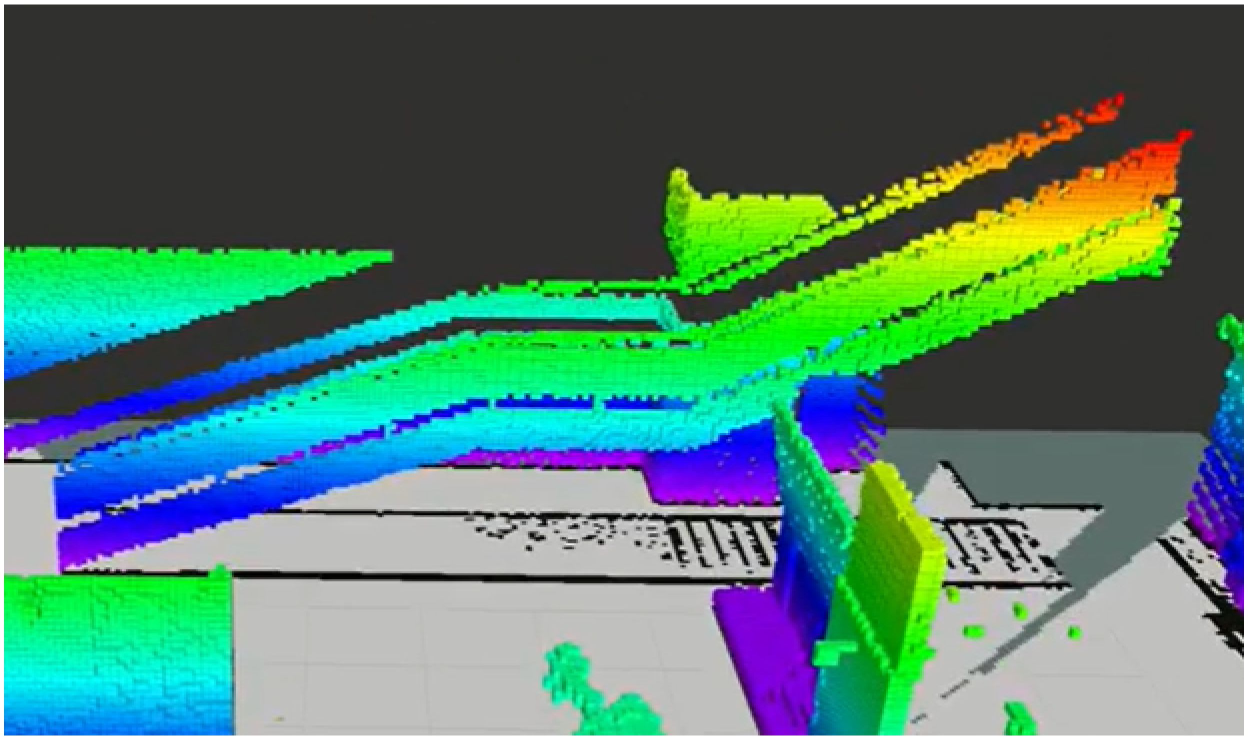

Figure 7: A Raised Staircase Being Observed as an Obstacle on the 2-D Map (black and white)

We also used the octomap_server to perform Z-Filtering on artifacts present on the ground of the simulated environment, and to eliminate noise, as it conveniently includes options for basic filtering.

Point Cloud Library

The Point Cloud Library (PCL) provides the necessary tools and functions for working with point cloud data types. We used PCL to transform individual clouds from the robot's coordinate frame to the global coordinate frame. While working with visual odometry, it was necessary to transform clouds that were viewed from the robot's frame into the global frame that would allow us to add them to the map. PCL's included transformation functions were useful for the set of nodes we created to handle this issue. However, after experimentation, it was determined that this process was already being done in TOUGH, resulting in these nodes being scrapped.

RGBD SLAM

The RGBD SLAM library allows for SLAM through image processing. The RGBD SLAM node takes in stereo images from a set of cameras, calculates their disparity, and pieces together a map. We attempted to use this process with Atlas's raw image data. Had the implementation been successful, Atlas could have performed SLAM very early into the project. However, the installation and application of this library to our project proved to be extremely troublesome and unreasonable, as it required extremely specific inputs.

Gazebo & RVIZ

Gazebo and RVIZ are applications designed to work with ROS. Gazebo is a physics-based simulation engine that allows models to traverse 3D environments. All of the simulation of this robot is done in Gazebo. Gazebo also allows users to alter terrain and place objects within the robot's environment.

RVIZ exists in order to visualize data and ROS-Topics. All of our team's 3D maps and 2D occupancy grids will be displayed in RVIZ, along with any point cloud or image topics we use to create them. RVIZ also provides the necessary tools for evaluating the effectiveness of our mapping algorithm.

Mapping Methods

This section will cover the methods we used to assemble maps of the robot's environment. All of our maps were created using sensor data from the stereo cameras and LiDAR.

Mapping with Viso2 Pose

Atlas can provide a point cloud of its environment from either the stereo cameras or the LiDAR. This point cloud is centered on the sensors. In order to combine point clouds into a map, they must be transformed to the same global origin. The transformation can be obtained by using the current 3D pose of the robot or using feature detection and matching between point clouds.

The gyroscope in Atlas' pelvis provides accurate 3D pose information for the pelvis of the robot. This pose can be obtained through the robot_state node in the tough_common package of the TOUGH library. Finding the pose of the stereo camera or LiDAR from the pelvis pose would require transforming the pose. The transformation would need to take into account the angle of the waist joint and the pitch of the stereo camera (for stereo camera point cloud). Errors would accumulate with each measurement, and the final pose might not be accurate enough. We did not know that the gyroscope pose could be used until much later in the project, and did not attempt this method of getting the robot pose.

Viso2, a visual odometry library, could find the 3D pose of the robot from the stereo camera images using feature detection. We used Viso2 to find the pose of the stereo cameras directly. Once the pose of the sensors is found, the point clouds can be transformed to the same global origin. Once transformed, the octomaps created from the point clouds can be compared and combined easily using a Bayesian filter.

We wrote and began testing code for mapping using Viso2.

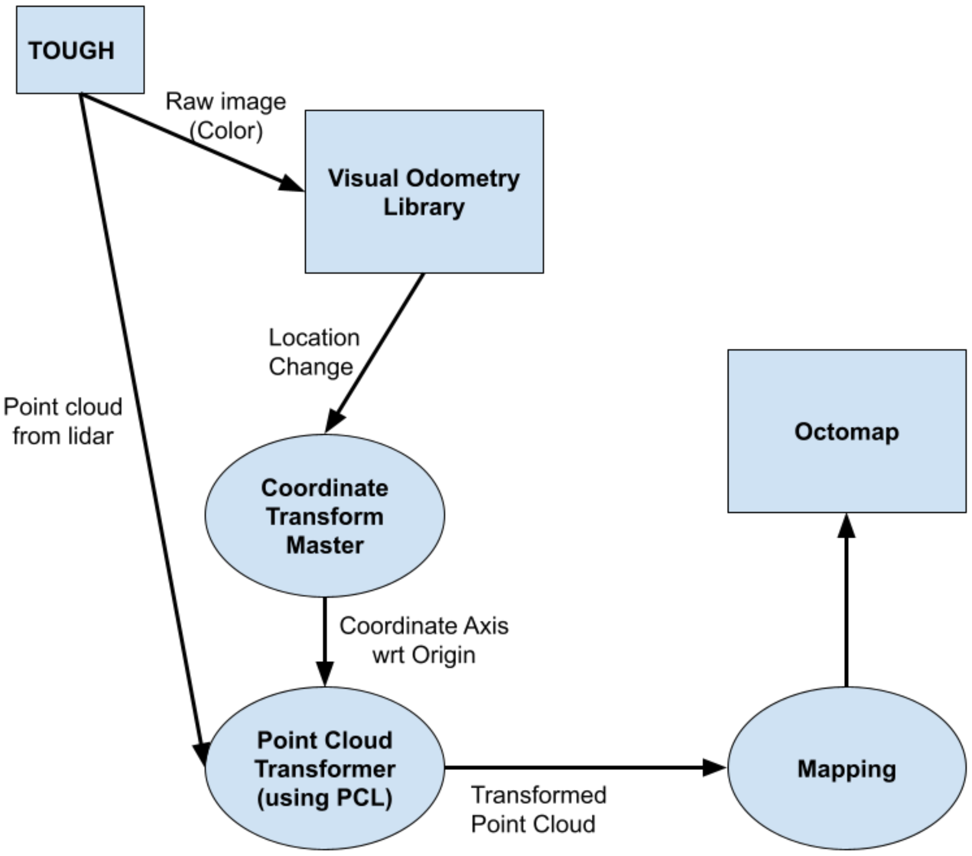

Figure 8: Visual Odometry and Point Cloud Transform Node Graph

We discovered that Octomap performs feature detection and mapping on point clouds and combines them into a larger map. In other words, passing raw point cloud data directly to octomap is sufficient to accomplish mapping. Making the first group of nodes created obsolete.

Octomap Mapping

Octomap uses feature detection and matching on point clouds to transform and match new point clouds to the existing map. It then uses a probabilistic algorithm to update the map. Mapping with octomap provides an accurate map of the environment as the robot moves and is capable of loop closure.

We remapped the published point cloud topic directly to the octomap server. The published octomap could then be viewed in RViz after downloading octomap plugins. By remapping the point cloud from the lidar to publish directly to the octomap_server, very accurate maps could be generated.

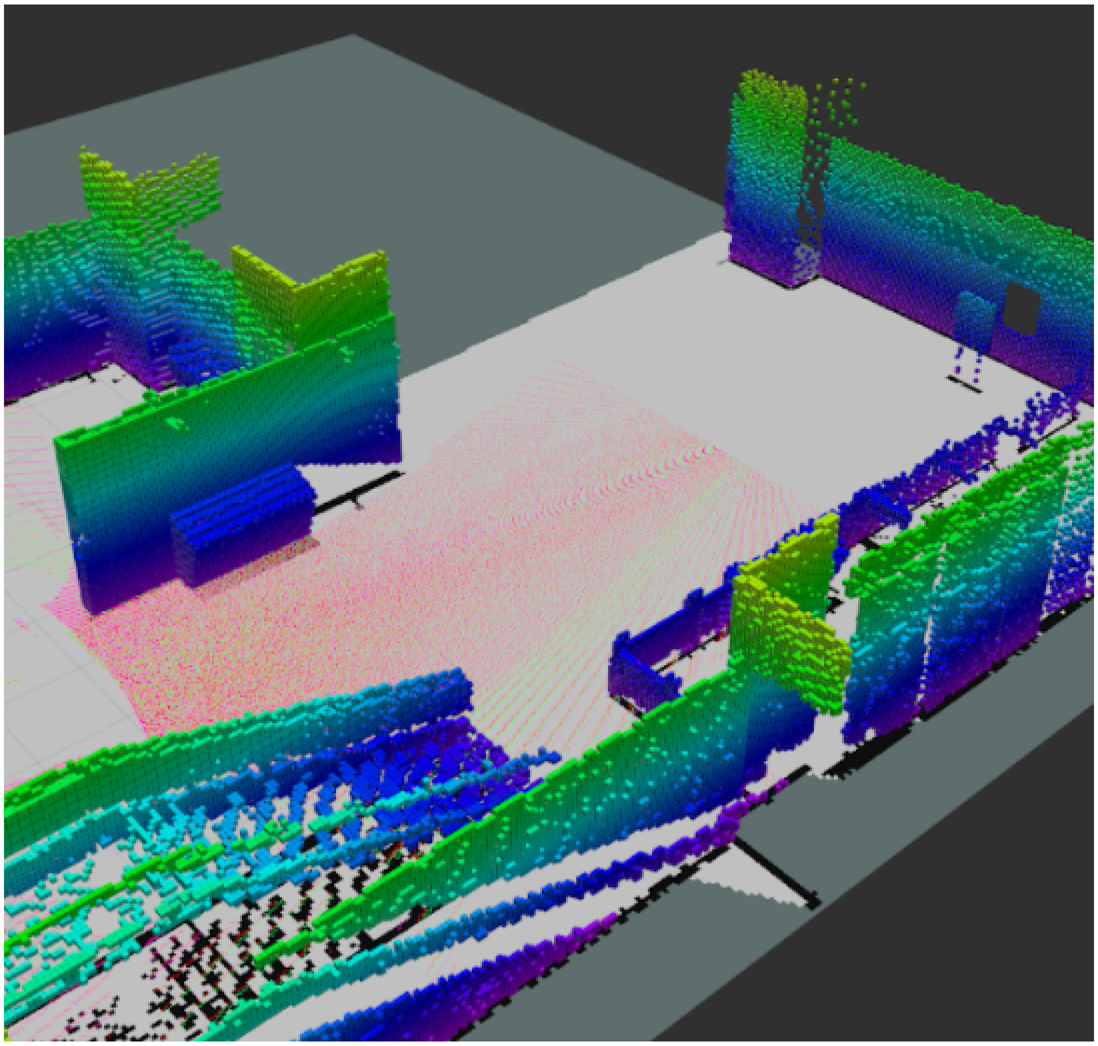

Figure 9: Octomap Generated by LiDAR Point Cloud

Stereo Camera vs LiDAR

We tested our mapping using point clouds from the stereo cameras, and point clouds from the lidar. The stereo camera had many artifacts, especially on the ground. We attempted to implement a z-filter to remove these artifacts from the point cloud. We were not aware that

Octomap had z-filter functionality built in, so we attempted to implement it ourselves by altering the pointCloud2 we received from the robot.

We used Viso2 to find the pose of the robot relative to the origin of the world and transformed the pointCloud2 data to this origin. We could then see if the point was lower than our minimum z value and transform the point down to the ground plane. Once z-filtered, the point cloud was transformed back to its original origin and passed to octomap. We did not finish debugging the program because we discovered that octomap could perform the z-filter.

After seeing the inferior quality of the stereo camera point cloud, we started experimenting with the LiDAR. The LiDAR must be started manually. First, the spindle speed of the LiDAR must be set using the command

```
rostopic pub /multisense/set_spindle_speed std_msgs/Float64 0.8
```

Next, the perception nodes must be launched using

```
roslaunch tough_perception_bringup field_laser_assembler.launch
```

To avoid excessive memory usage, the LiDAR will stop after publishing 5 maps. Each map takes a minimum of 8 seconds in real time to publish, which can be upwards of 30 seconds in simulation with a low real time factor. This command restarts the LiDAR:

```
rostopic pub /atlas/pause_pointcloud std_msgs/Bool false
```

The LiDAR point cloud had no artifacts, so we did not perform any filtering to improve the map.

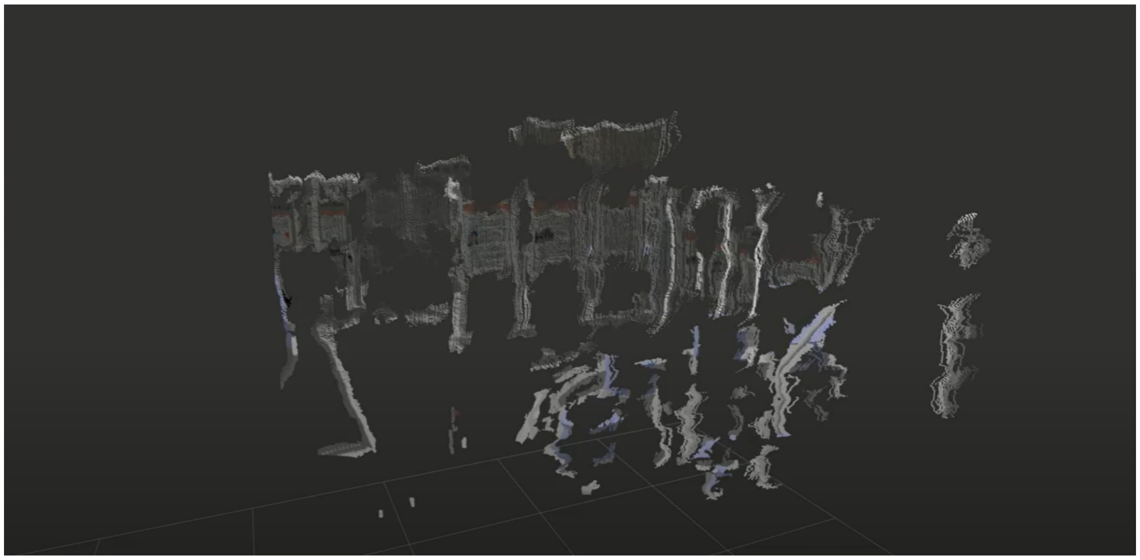

Figure 10: Stereo Camera Point Cloud in rviz

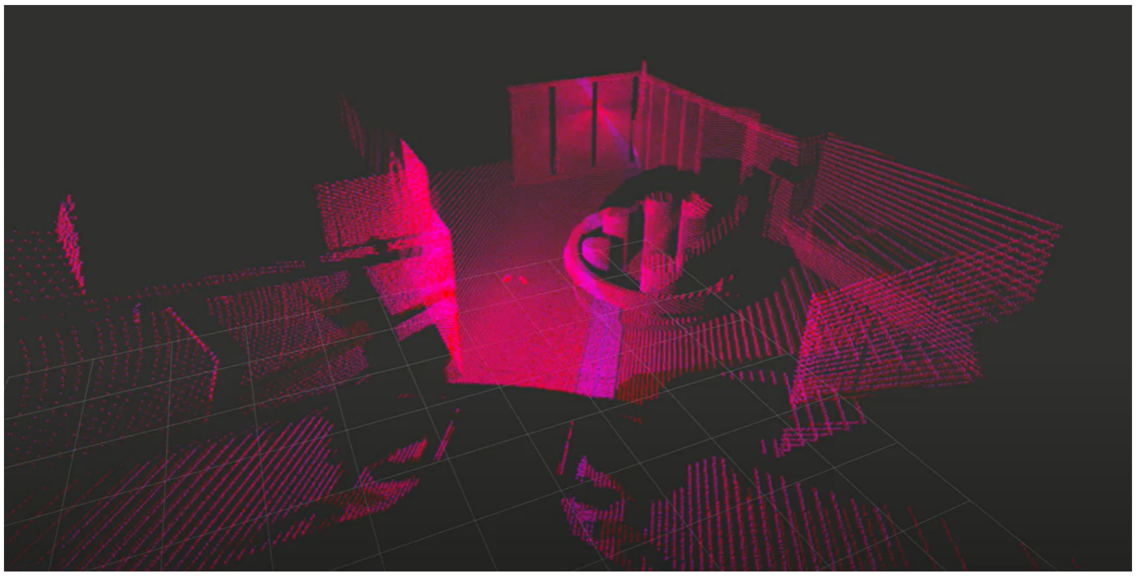

Figure 11: LiDAR Point Cloud in rviz

Frontier Based Exploration

This section will cover the basis for which we organized our frontier based exploration algorithms. Frontier based exploration defines how the robot selects areas of the map to explore, and how our programs discern the difference between unknown and known territories of the environment.

Exploration Structure

Upon developing the ability to accurately map the robot's surroundings, it was necessary to implement autonomous exploration of the area. Through mapping, we were able to generate both two and three dimensional maps of the surroundings. The two dimensional map was a projection of the three dimensional one, meaning areas that inclined enough above the ground at those with overhang would be marked as impassable, meaning it was optimal for flat environments such as indoors or paved locations, but would struggle in highly uneven terrain. However, as the robot is primarily tested by walking on flat ground, the two dimensional map was selected for use in exploration.

As a process, the robot would begin by mapping all currently visible areas with the LiDAR, generating a gridded 2D map of known passable and impassable areas, shown as light gray and black squares in the map below, respectively.

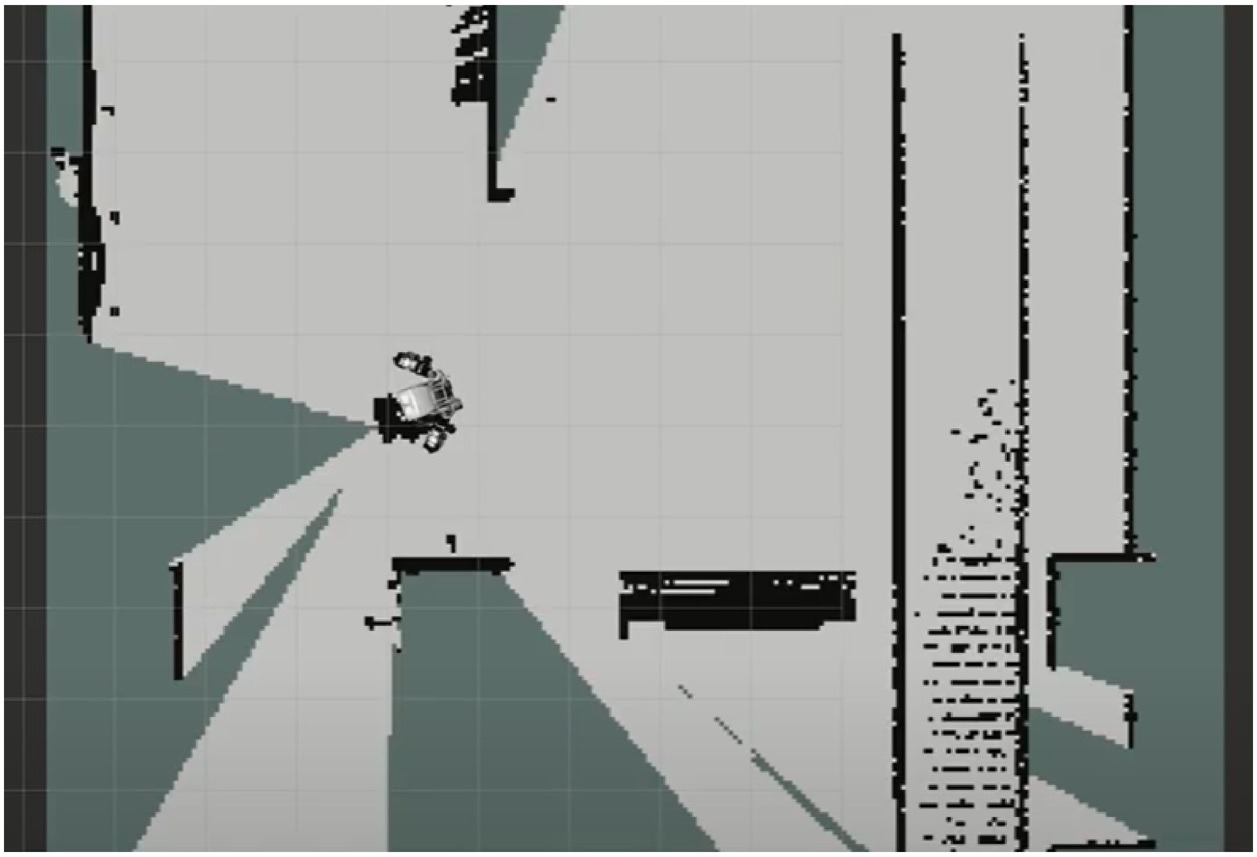

Figure 12: Atlas Walking in a 2-D Map

The known areas that border the blank edges of the map, or unknown spaces, were considered to be “frontiers.” These frontier grid cells would form the basis for the exploration. By determining the closest frontier and moving there, the robot could explore unexplored areas. Doing this repeatedly until there are no remaining frontiers to explore forms a complete and fully explored map.

To do this, several ROS nodes were created to work in conjunction with the existing mapping code. To determine and choose from the frontiers, the FrontierExplorerNode was created. It also calculates the centroid of the closest frontier, which is the desired point to navigate to. The CSpaceNode is used to buffer the area taken up by the impassable spaces on the map to ensure the robot does not collide with any obstacles. The final written node, the NavigationNode decides where the closest reachable known point is to the desired centroid and sends this information to the TOUGH footstep planner.

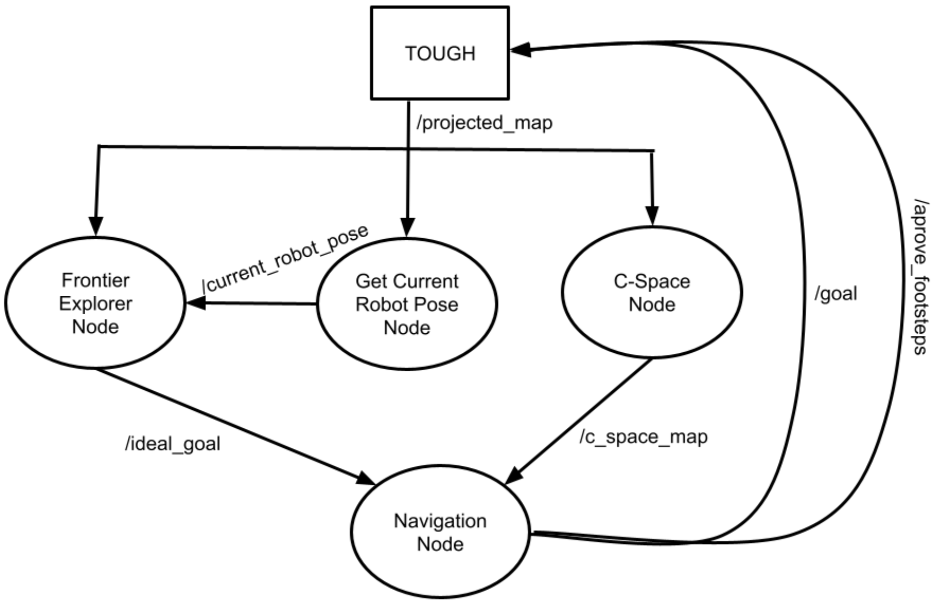

Figure 13: Frontier Based Exploration Node Graph

FrontierExplorerNode

We began the FrontierExplorerNode with a similar process as was used for the various mapping nodes. We initially created a C++ template node that could listen to and publish topics and filled it in by creating various functions to complete the tasks necessary for creating and analyzing frontiers. The frontiers were created by reading over a ROS occupancy grid and adding the squares located next to unknown regions into a list of frontiers. This list was a vector data type containing vectors of coordinate pairs, which were arrays of length 2. The two vectors could be of variable length depending on the amount of frontiers and the length of the individual frontiers listed within. This and other C++ related errors proved to be a large inconvenience. Fortunately, we were able to utilize ROS's capability to use both C++ and Python code. Because the nodes programmed in either C++ or Python communicate through rostopics, the arithmetic being done inside the node can be done in either language. For the FrontierExplorerNode in specific, Python is very well suited to editing and reorganizing lists of data. The new PythonFrontierExplorerNode node was created very similarly to the C++ version, albeit following the different syntax for setting up a publisher and a subscriber. The node subscribes to the OccupancyGrid (2D map) created by the octomap_server and publishes a point ROS message that represents the desired frontier centroid to be navigated to. To successfully go from map data to a navigation goal, there are several tasks that must be done. To start, the functions found the frontiers by checking over the entire 2D map and formulating lists of grid cells (stored as tuples of x and y coordinate values) that were adjacent to both an unknown space and another frontier space. If a point was adjacent to an unknown space, but not another frontier, it would be stored as a single point frontier. These lists of adjacent frontier points were themselves stored within a list of frontiers. The centroid for each individual frontier from this list of lists of coordinates is then calculated by averaging the x and y value of each coordinate within the frontier. Once a list of centroid is determined, the closest one to the current robot pose is selected as the desired navigation goal for continued exploration by calculating the euclidean distance to each and selecting the lowest value. Additionally, the frontier explorer publishes a false boolean value, which the LiDAR running through TOUGH subscribes to. This ensures that the LiDAR continuously updates while the robot navigates through the map.

CSpaceNode

The CSpaceNode operates along with the FrontierExplorerNode. It subscribes to the same occupancy grid as the frontier explorer and publishes an edited occupancy grid. However, this occupancy grid includes “padded” obstacles, where the size of the impassable areas has been increased so that the risk of collision is reduced significantly. It works fairly simply, by iterating through the entire grid, checking for occupied cells. Each cell contains a value from 1 to 100 indicating the possibility that the cell is an obstacle. If this value is over 50, the CSpaceNode changes it to 100 to confirm that it is an obstacle. It then changes the values of the surrounding cells to 100 to pad them. The amount of cells that are padded around the obstacles are controlled by an integer variable that can be changed for optimization. This node was also written in Python because it also involves list operations, therefore, it’s file is named PythonCSpaceNode

NavigationNode

The PythonNavigationNode is the final node in the frontier based exploration process. It subscribes to the padded map from the CSpaceNode and the navigation goal from the FrontierExplorerNode. It then checks if the location of the navigation goal is within a walkable location. If it is, this point is sent to the TOUGH footstep planner to generate a path to follow and to actuate the robot along it. If it is not a walkable location, the navigation node calculates the closest walkable location to the goal, again through euclidean distance, and sends this point to the footstep planner. This was meant to be called repeatedly until the robot has traversed and mapped all of the reachable frontiers. Once there are no more, and the map has been completed, the process stops, resulting in a “closed-loop map.”

Pathfinding

Pathfinding was handled entirely through the TOUGH footstep planner. The custom node detailed above sends the footstep planner the desired navigation goal. Upon receiving this information, the footstep planner can generate a sequence of steps for the robot to follow that will result in the robot reaching the goal without stepping too far, colliding with its surroundings, or losing balance.

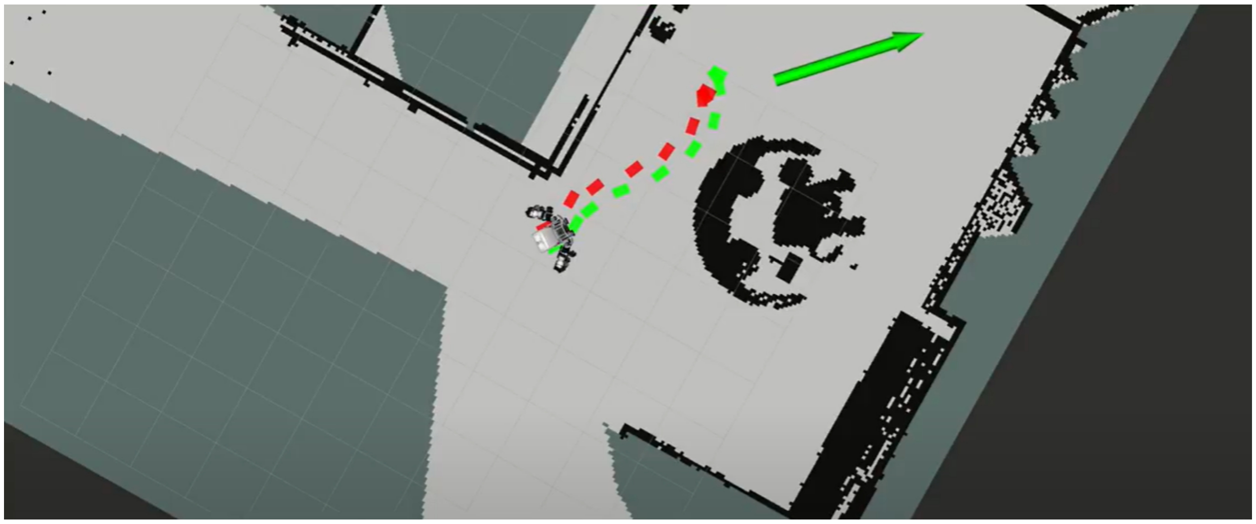

Figure 14: TOUGH Footstep Planner Generating a Path of Footsteps

Then, the robot is actuated along these steps, also within TOUGH. The planner also requires a separate topic to be published that approves the steps generated. The implementation for this was started, but not finished, meaning that the navigation goal generated by the nodes needed to be input to the planner manually, using the TOUGH GUI. Because of this, observing the generated map by eye and clicking on a goal in the GUI also works for exploration.

Efficient Code Execution and Testing

To allow for efficient testing of our algorithm, we initially designed a simulated environment imitating a coffee shop. This had lots of obstacles to be mapped, however there were not many walls and corridors, which we believed would be helpful for testing effectively. To do this, another simulated environment was created in the same way as the previous one. This involved placing obstacles in Gazebo from its library of pre-created items. We found a useful model in the form of the IEEE ISCAS museum in South Korea.

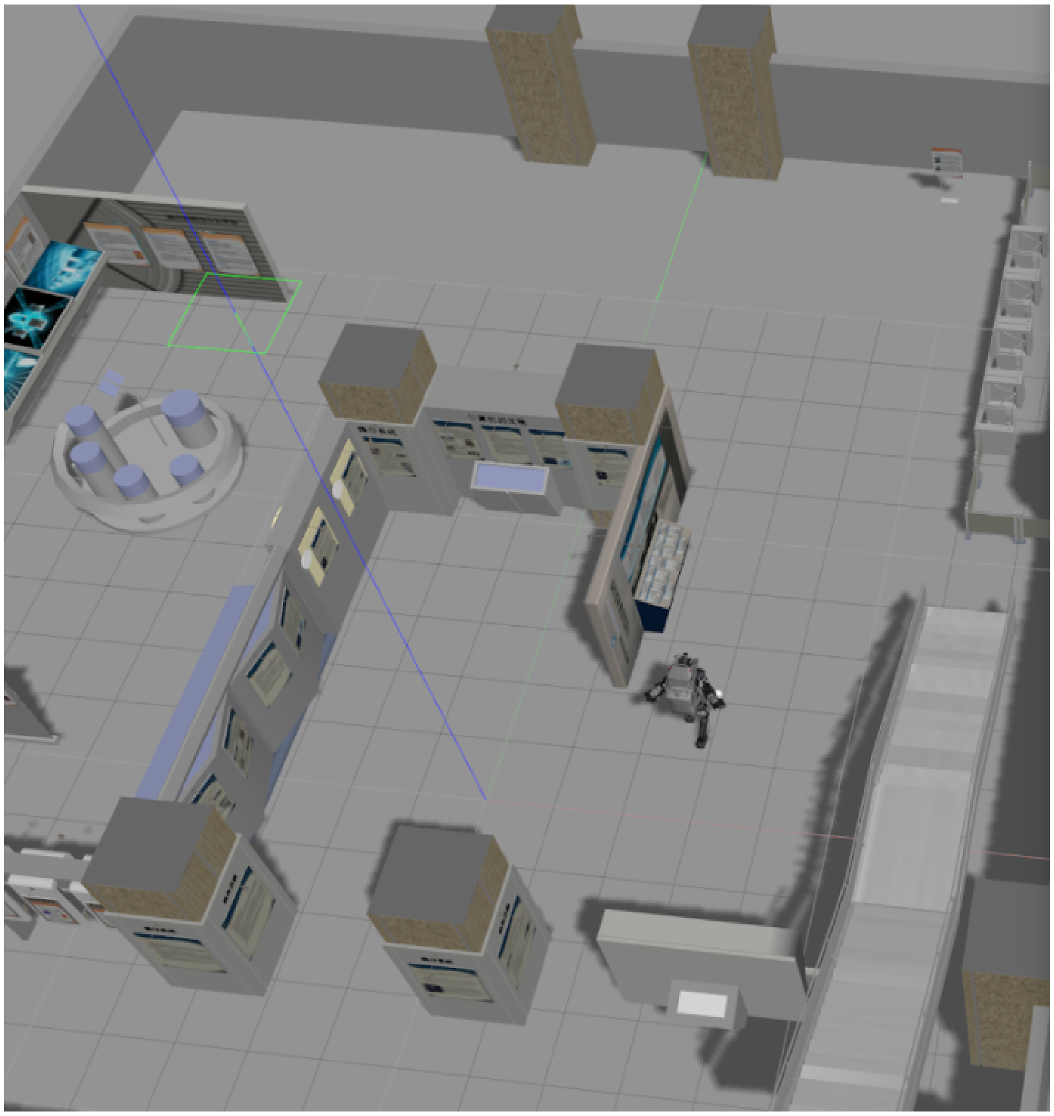

Figure 15: Atlas Walking in our Simulated Environment

This had all of the features we desired, including mostly flat floors, some traversable 3-D surfaces. And walls forming corridors to be explored. Creating the world file also required spawning the atlas robot within the environment. To do this, we made sure that the museum model was oriented on the origin, where atlas could walk as it does in the empty default TOUGH simulation. We duplicated the code pertaining to spawning Atlas from the default world file into our new one.

In order to ensure the code runs properly, there are several steps that must be done. Upon completing a node in C++, it must be added to the CMakeList file for the desired package.

This will create an executable file. To make executable files for nodes created in Python, running the chmod command on the node in their directory in a terminal makes the file itself executable. These files, as well as other important executable tasks to be run, can be put together into a launch file that will run all of the nodes at once. This was significantly more convenient than running each node and each other process separately. We created several launch files, but our main launch file, named slamnodes.launch, runs the Gazebo simulation with Atlas in our environment, the octomap server, and our nodes. A similar file was created for the obsolete visual odometry nodes, but that is not used. The slamnodes file allowed us to conveniently run all of the necessary nodes for testing to generate results.

Results

Our final SLAM algorithm could perform closed loop mapping with no noticeable drifting in the map as the robot moved. Using the LiDAR, the only artifact in the final map was the part of the robot's head, which the LiDAR could see. No artifacts were on the ground, walls, or other objects in the robot's field of view.

Comparison of LiDAR and Stereo Camera Performance

We initially used the point cloud from the stereo camera to create the map. The stereo camera point cloud has several issues. Shadows in the environment were mapped as 1-3 grid square high obstacles. These shadows were down projected as obstacles on the 2D occupancy grid, and so were treated as an obstacle when the robot was walking around.

The stereo camera point cloud also added a lot of artifacts. The artifacts appeared on walls and made them appear bumpy instead of flat. Sometimes, there would be large spikes coming out of surfaces. The stereo camera point cloud also had a lot of artifacts in front of the robot. There would be small 1-5 block clumps of "obstacles" placed randomly in front of the robot. These wouldn't disappear and more would appear as the robot moved around, resulting in larger and larger clouds of imagined obstacles as the robot continued to move. These artifacts also prevented the robot from path finding correctly since they were also downprojected as obstacles on the 2D occupancy grid.

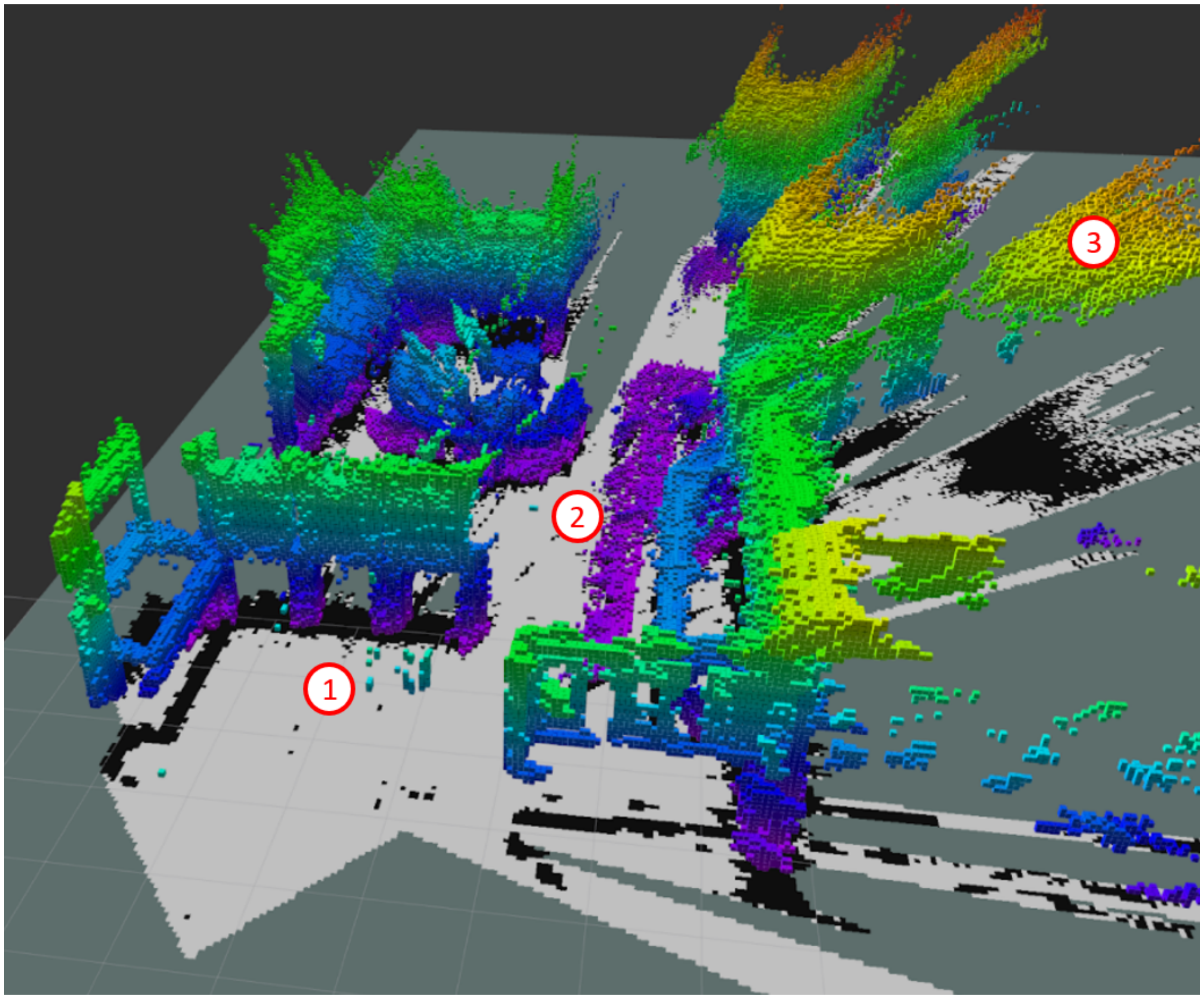

Figure 16: 3-D Map Resulting from Stereo Cameras with both Motion (1) and Shadow (2,3) Artifacts

The LiDAR point cloud takes long and is more memory intensive to calculate, but it is of significantly better quality than the stereo camera point cloud. The LiDAR point cloud has no artifacts and produces clean edges on all obstacles. The only artifact that appears is a block on the robot’s head when the LiDAR is started. The head is not detected by the LiDAR after the robot has moved. This could be fixed by filtering out the obstacles within some radius of the robot’s initial position.

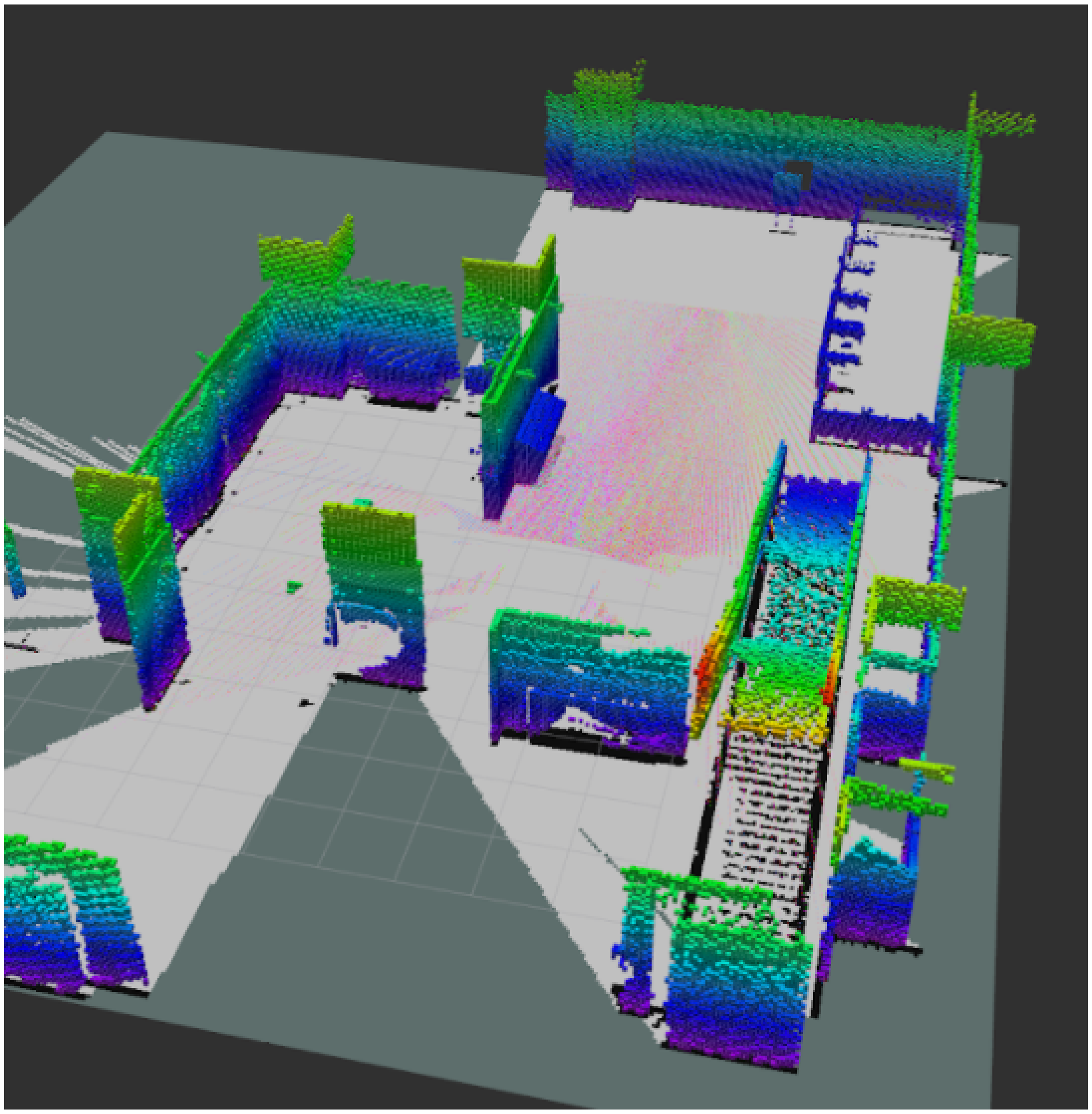

Figure 17: 3-D Map Resulting from Lidar with Updating Point Cloud Data Visible

Map Completion

As detailed in the methodology, the robot is able to create closed loop maps with the aid of manual approval of steps using the TOUGH GUI. By doing this, we were able to create a full map of our simulated environment, the IEEE ISCAS museum. This process was done using the LiDAR data because of its aforementioned accuracy.

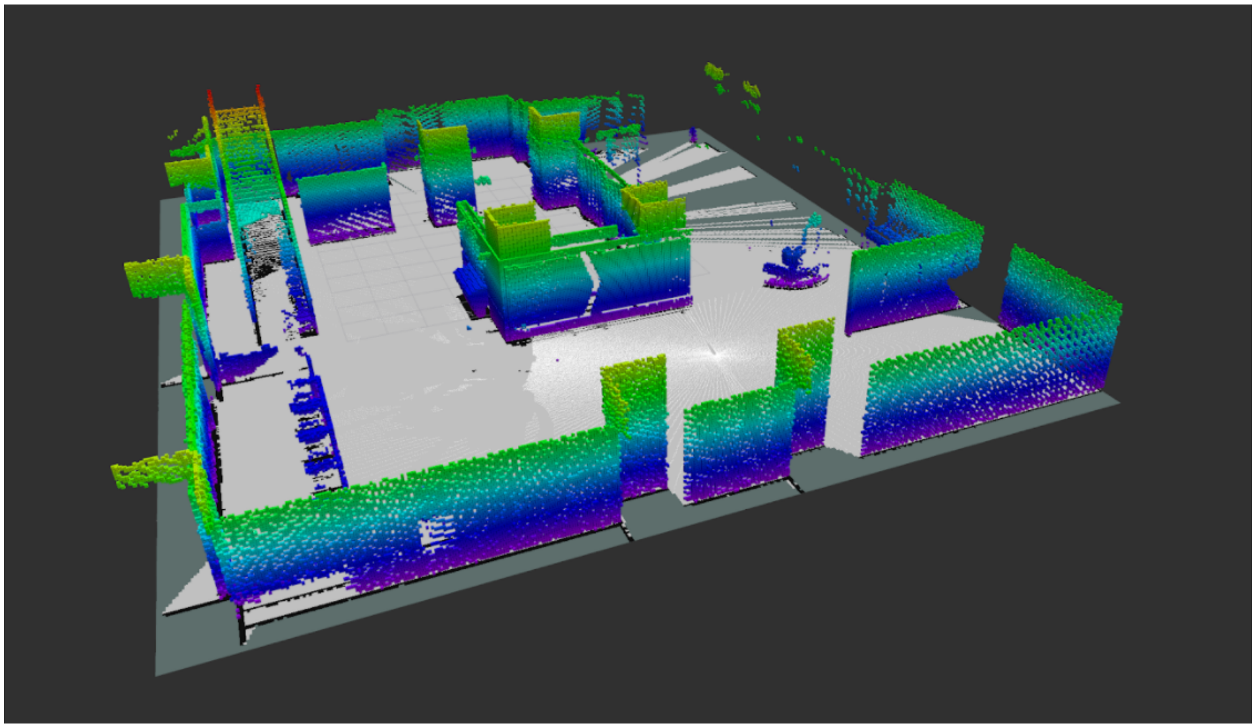

Figure 18: 3-D Map of Simulated Environment using LiDAR

This meant that the process of completing a closed-loop map was very slow because of the LiDAR's speed. In fact, the simulation ran so slowly, that it was necessary to switch computers from a relatively powerful laptop to a more powerful desktop to continue testing, even when only using the stereo cameras. The real time factor displayed in Gazebo was typically around 0.05 (one twentieth of a second per second) for the more powerful setup using LiDAR data. Because of this, high computer specifications are necessary when generating maps in this way. The slow simulation speed resulted in closed maps of the museum that took about 40 minutes to complete. This time scales with the size of the environment. Nonetheless, the LiDAR was able to create a very accurate 3-D map of the environment. Because of the vertical walls and even terrain of the environment, the 2-D map generated by the octomap_server was also very accurate, with well defined traversable areas and obstacles. Closed-loop mapping was not attempted with the stereo cameras due to the large amount of artifacts they generated during operation.

Troubleshooting and Autonomous Frontier Exploration

We were not able to complete debugging of the autonomous frontier exploration due to an issue with the IHMC controllers. Throughout the project, we would run into issues with the IHMC controllers preventing the project from making or preventing the simulation from starting (more often the latter). Even though we were supposed to be running everything locally, the IHMC controllers would still be updated from the external repository. This would cause the installations on all of our computers to break simultaneously. We ran into this issue while we were working on our autonomous frontier exploration and were not able to resolve it.

Using ROS Kinetic also caused issues. There are many libraries and codebases that may work, but have not been updated over really long periods and would cause issues when trying to run in conjunction with TOUGH and our other components in our Catkin workspaces. We also found libraries which only worked on newer versions of Ubuntu or ROS. ROS Kinetic also became unsupported near the end of our project. This may have been why the IHMC controllers stopped working. We expect any future projects to struggle to set up and run ROS Kinetic, TOUGH, the IHMC controllers, and any other applications or libraries in the project.

We also ran into multiple other issues with TOUGH and related libraries. One issue that we ran into was that gazebo could not be launched when we used the default branches. A new branch was created that allowed us to use Gazebo. Unfortunately, installing other necessary files would change the branch that we were on. We ended up with a “loop” of errors where fixing one library would break the other. Instructions on fixing this error are found in Appendix B.

Another library that we had trouble with was RGBDSLAM. Even though many people appear to use this library for kinects, we had many issues related to installing and using the library. We found misplaced files, local file paths that should have been global, and crashing issues related to launching the GUI. We were able to run and collect data in the version without a gui, but the point cloud was completely misshapen. Since we had only managed to get the library working on a single computer, we decided to try a different approach.

Another issue we had was with graphics drivers. TOUGH cannot run using a virtualized GPU (and thus also can’t run on a VM). It will crash at some point while the simulation is launching. One of our computers did not have graphics drivers available for Ubuntu, or the installed drivers weren’t being used by the system. Despite this, the simulation ran consistently until that computer crashed and Ubuntu had to be reinstalled. The simulation never ran again on that machine. On another machine, the graphics drivers would stop working on a regular basis. When the computer was started up, it would be stuck in a loop at login. The UI had to be stopped and graphics drivers reinstalled before the machine could work again.

The simulation also required a lot of processing power in general, and would regularly crash computers running it. These crashes would sometimes corrupt files and result in the reinstallation of Ubuntu.

Discussion and Analysis

The results detailed above provide important revelations about implementing SLAM on a humanoid robot. By collecting and analyzing the data, we were able to understand the successes and shortcomings of our methods. The work we did in the process has set a basis for future undergraduate work with the Atlas robot at WPI, but there are certain caveats to continued work. Overall, our results provide valuable insights for the future of humanoid robotics research, particularly in WHRL.

Successes and Shortcomings

Our final result was successful at implementing SLAM, but it was not without tradeoffs. One tradeoff was using the LiDAR instead of the stereo cameras. As previously mentioned, the LiDAR was significantly slower than the stereo camera. The extremely low real-time factor would have most likely translated into a very slow operation on the physical robot, had testing in the lab been feasible. However, the speed did not end up being a detrimental issue because the range of the LiDAR was very long and the environment featured multiple wide-open spaces. The LiDAR point cloud data would be published before the robot could reach the frontier of the mapped area. Had we attempted to use the stereo cameras to complete the map, the result would have been far too messy to understand, let alone to use as a map for navigation. The walls of the environment might have been somewhat clear, but the noise artifacts created by the robot's motion would have likely filled up the entire area in which the robot had moved. Combined with the artifacts created by shadows, the stereo camera mmp would not have been useful, especially if the robot needed to navigate through a C-space map.

Some other shortcomings were more noticeable, however. The downprojected map was not ideal. Any obstacles of artifacts in the 3D map, regardless of their size or heights, would be mapped to an obstacle in the 2D map. If there had been a rug on the ground or a ceiling above, an obstacle would have been registered, even if the robot was fully capable of walking on or over it. There is not enough information in the downprojected map to represent the 3D space accurately. This could be partially mitigated by adjusting the z-filtering values for the octomap_server, but this is not a permanent solution. Ideally, the algorithm would be able to understand which obstacles could be walked on, such as floors on uneven ground. However, these changes would make for a 3-D SLAM algorithm, while this project implemented a 2-D algorithm.

The footstep planner would plan the path of the robot to a goal, but it also had its limits. Goals could be inputted manually or by publishing to a topic, but either way had to be approved before the robot would walk. The footstep planner also did not appear to create a large enough c-space of the map before path planning, and the robot would occasionally hit corners narrowly, which caused it to fall. Because the autonomous coordination with the footstep planner was incomplete, the CSpaceNode and NavigationNode were not used to generate closed-loop maps and were not able to be tested as thoroughly as the rest of the application. However, if the coordination with the footstep planner was completed, they could have provided very useful methods for avoiding these rare collisions.

Ultimately, our 3D map was accurate and clean, but the other tools we used should be improved before they are used as a robust autonomous exploration algorithm, for either simulation or the physical robot. Working with the TOUGH libraries was one of the most difficult tasks we approached. Enabling the algorithm would require this through tasks like creating and approving steps autonomously, providing TOUGH with the padded C-space map, and more in-depth use of the 3-D maps.

Significance

The results of this project carry a certain level of significance. To begin, this is the first known implementation of SLAM on the Atlas robot, and one of the few implementations on humanoid robots. As novel research, we helped to expand the understanding of using broadly applicable methods on humanoids, specifically at WPI. In doing so, we were able to test different methods that make SLAM difficult for humanoids. We discovered the possibility, but impracticality, of using a 2-D algorithm for a robot capable of 3-D actuation. Additionally, in our literature review we came across previous research that struggled with optimizing performance and speed, which was an issue we confirmed to be present in implementing slam. Additionally, we were able to define specific tasks that the two multisense cameras, LiDAr and stereo cameras, are more applicable to through testing them in the simulated environment.

Because of the unique timeframe located entirely within the COVID pandemic, we were able to find useful ways of operating within simulation and out of the lab. By setting up the TOUGH environment on our own computers, we followed a process that could be used to enable remote development of humanoid robotics research. The project also helped to open up new opportunities for research in WHRL. The point we left off our research has many possible paths for future work.

Future Work

The completion of this project opens the possibility for future work, both as an extension of this project specifically, and as tangentially related work. To begin, implementing our code on the actual robot would be a clear step forward and an important one to prove the validity of our algorithm. While meeting with several people in the lab to run the robot was not possible this year due to COVID, this will likely change in the future. Additionally, completing autonomous frontier exploration was a goal of the project that we were not able to reach. Finishing this is another very clear possibility for continued work. However, the degradation of ROS Kinetic, in which this project was created, makes a direct continuation of the project impractical, for the time being. Because of this, the most practical step forward for continuing work is to migrate the TOUGH control libraries to a more recent ROS distribution, such as melodic or noetic. This would require large amounts of both effort and knowledge of TOUGH to do, but it is essential for continuing work with TOUGH.

Other possible paths for continued work depend on updating TOUGH. For instance, improving the SLAM algorithm created in this project could be a practical way to increase the robots capability to interact with novel environments. An example of how this could be done is by optimizing how frontiers are selected for exploration. In our desired implementation, the robot proceeds to the closest frontier, but this could, hypothetically, not be the optimal choice. Perhaps a study could be done to determine if a combination of factors, such as size and distance could be more optimal for quick closed-loop mapping.

Other possible routes of future work are less directly related to SLAM on Atlas, but are nonetheless just as viable. Moxe involve interacting more with the generated 3D maps. The stereo camera point clouds could be used to interact with objects because they maintain color data. This could be useful in retrieving an item, like a red first aid kit for example. This would be a more reasonable application for the stereo cameras than SLAM, as we found the LiDAR more useful in that case. Additionally, the robot only interacted with down-projected 2-D maps in this project, but the 3-D maps provide a lot of useful data. They could be used for 3-D obstacle avoidance or navigating uneven terrain, which was not done in this study.

Conclusion and Recommendations

By the end of our testing we were getting fairly confident maps out of our SLAM implementation. Our maps were simply evaluated visually against the Gazebo environment that simulated the IEEE ISCAS museum in Korea. We chose this environment because it provided a balance of walls, spaces, and features which were excellent for viewing the results of our sensor data. The simulation also provided us with a flat walkable terrain, which provided the best case scenario for TOUGH's footstep planner.

Ultimately our results proved that 3 dimensional LiDAR scans, when used to update a 3D occupancy grid, can provide us with a working methodology for performing SLAM on a humanoid robot. The robot navigates these maps in a global coordinate frame, effectively localizing itself within the maps it's created. That being said LiDAR, though proven very reliable in simulation, still has some major drawbacks. It takes almost 30 seconds in simulation for the LiDAR point cloud to populate the map. If this method was ever used on a faster or more agile robot, the speed of the robot may supersede that of the map updating. This is generally not good practice, as the robot is already in an unknown frontier when it begins mapping again, resulting in inconsistencies in localization.

The stereo cameras have proven much more efficient in this regard, and it may be advisable to use them in fusion with the LiDAR for more confidence. This confidence would prove crucial if Atlas were to navigate uneven terrain or rough obstacles. In those cases it's imperative to have a good understanding of the robot's surroundings so that it can best plan its footsteps.

Other future work on this project could also incorporate better filtering techniques for the stereo point clouds. We attempted the implementation of a Z-Filter but found it wasn't quite great at eliminating artifacts that appear from shadows. It would be advisable to instead use a filter that eliminates individual points in the cloud that aren't part of a large cluster. This would combat the appearance of artifacts that ultimately provide the robot with false obstacles. These features could make the difference between a good mapping algorithm and an excellent one. It is because of this that we recommend continuing research for SLAM on Atlas if possible. There are many possible ways to improve our algorithm and developing and entirely 3-D functionality is definitely possible.

Currently the Atlas robot can operate in a closed environment within a simulation, but it would have been interesting to perform more testing on the physical robots. We would assume that the behavior would be similar to that of simulation, but there are plenty of variables that likely influence this reality. The robot's lidar scan may very well be significantly less accurate in reality than in simulation. Additionally it's possible that our footstep planner (which assumes level ground) may struggle with an actual floor.

It’s being able to operate in any environment that would make this implementation incredibly valuable. We would hope that with enough progress on this topic would eventually result in the robot being able to map truly novel environments such as a forest or the site of an accident. The map data it creates in these areas could prove to be incredibly valuable to crisis response or surveillance teams. Humanoid robots will continue to be developed and changed until they can completely replace humans athletically wherever needed. It’s research like this that our team hopes will continue to pave the way to the future.

Bibliography

Temporary to keep track of citations:

[1] https://openslam-org.github.io/gmapping.html

[2] https://people.eecs.berkeley.edu/~pabbeel/cs287-fa11/slides/gmapping.pdf

[3] https://arxiv.org/pdf/1301.3853.pdf

[4] https://ieeexplore-ieee-org.ezpxy-web-p-u01.wpi.edu/document/5649751

[5]https://towardsdatascience.com/wtf-is-sensor-fusion-part-2-the-good-old-kalman-filter-3642f321440

[6]
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.478.4480&rep=rep1&type=pdf

[7] https://ieeexplore-ieee-org.ezpxy-web-p-u01.wpi.edu/document/5649751

[8] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3871090/

[9] https://link-springer-com.ezpxy-web-p-u01.wpi.edu/article/10.1007/s11370-017-0230-0

[10] https://fjp.at/posts/ros/ros-kalman-filter/

[11]https://www-sciencedirect-com.ezpxy-web-p-u01.wpi.edu/science/article/pii/S0957415818301338

[12]http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#:~:text=A*%20is%20the%20most%20popular,a%20heuristic%20to%20guide%20itself.

[13] http://www.cs.columbia.edu/~allen/S19/Student_Papers/chestnutt_laser.pdf

[14] Analysis of Flat Terrain for the Atlas Robot

[15] Deep Q-Learning for Humanoid Walking

[16] http://files.carnegierobotics.com/products/MultiSense_SL/MultiSense_SL_brochure.pdf

[17] Boston Dynamics service bulletin
https://www.dropbox.com/sh/i55edrserg4fyvk/AAABRcX5JEzKExUd7iqQF5Eea/Atlas%20Service%20Bulletin?dl=0&preview=ATLAS-01-0021_Atlas_Robot_Service_Bulletin_V1.docx&subfolder_nav_tracking=1

[18] https://arxiv.org/pdf/1607.08089.pdf (partial footholds and line contacts)

[19] https://arxiv.org/pdf/2004.12083.pdf (large step ups)

[20]
https://www.researchgate.net/publication/3930145_The_3D_linear_inverted_pendulum_model_A_simple_modeling_for_a_biped_walking_pattern_generation

[21]http://docs.carnegierobotics.com/S7/api.html

[22] http://wiki.ros.org/kinetic/Installation/Ubuntu

Appendices

Appendix A: Getting Atlas in Gazebo

First, follow the install instructions in the TOUGH repo, then do this:

cd ~/catkin_ws/src

wget
https://raw.githubusercontent.com/WPI-Humanoid-Robotics-Lab/atlas_workspace/master/atlas_gazebo_rbe595_ws.yaml

vcs-import < atlas_gazebo_rbe595_ws.yaml

cd ~/catkin_ws/ && catkin_make install

source ~/catkin_ws/install/share/drcsim/setup.sh

# Download the bash file in the drive or copy to gazebo_possible_fixes.bash from below, then run:

```
bash gazebo_possible_fixes.bash

source ~/catkin_ws/install/setup.bash

source ~/catkin_ws/install/share/drcsim/setup.sh
```

#launch!
roslaunch ihmc_atlas_ros ihmc_atlas_gazebo.launch use_local_build:=true

gazebo_possible_fixes.bash file contents:

```
#!/bin/bash

ihmc_repo_path=~/repository-group
catkin_path=~/catkin_ws
ihmc_path=${catkin_path}/src/ihmc_repos

echo "Git Pulling ihmc repos"
cd ${ihmc_path}/ihmc_ros_core && git reset --hard && git checkout
open-robotics-0.11.0-release && git pull
cd ${ihmc_path}/ihmc_ros_control && git reset --hard && git checkout
open-robotics-0.11.0-release && git pull
```

```
cd ${ihmc_path}/ihmc_atlas_ros && git reset --hard && git checkout
open-robotics-0.11.0-release && git pull
cd ${ihmc_path}/ihmc_gazebo && git reset --hard && git checkout
open-robotics-0.11.0-release && git pull

echo "catkin_make install"
cd ${catkin_path} && catkin_make install
source ${catkin_path}/install/setup.bash
source ${catkin_path}/install/share/drcsim/setup.sh

echo "ihmc repo branch setting"
cd ${ihmc_repo_path} && rm -rf ihmc-open-robotics-software && git clone
https://github.com/WPI-Humanoid-Robotics-Lab/ihmc-open-robotics-software.git
--branch gazebo_devel
${ihmc_repo_path}/ihmc-open-robotics-software/gradlew
```

Appendix B: How to fix Bugs (some of them, anyway)

Catkin_make bugs

- if your catkin_make is failing with /usr/bin/ld: cannot find -lvtkproj4 (or similar) run this with the name of your file substituted in:

```
ln -s /usr/lib/x86_64-linux-gnu/libvtkCommonCore-6.2.so
/usr/lib/libvtkproj4.so
```

Gazebo Bugs

If you get an error with robot description not found or are able to catkin_make but gazebo doesn't run, you may be in the error loop.

Or maybe not. Who knows. Here is how you might be able to get out of it:

Before starting, delete the build and devel folders in your catkin_ws and make sure you are on the kinetic-0.11.0 branch of tough. Catkin_make, then start the instructions below.

Run these:

```
cd ~/catkin_ws/src
```

```
wget
https://raw.githubusercontent.com/WPI-Humanoid-Robotics-Lab/atlas_workspace/m
aster/atlas_gazebo_rbe595_ws.yaml

vcs-import < atlas_gazebo_rbe595_ws.yaml
```

You are now in the RBE595 branch of tough
Go to tough and change back to kinetic-0.11.0
Download this entire folder and put it in tough/tough_control
`catkin_make`
Gazebo should still not work
Run this:

```
source ~/catkin_ws/install/share/drcsim/setup.sh
```

Running command below should still throw an error related to install/share

```
roslaunch ihmc_repos/ihmc_atlas_ros/launch/ihmc_atlas_gazebo.launch
```

Copy the launch folder from catkin_ws/src/ihmc_repos/ihmc_gazebo into catkin_ws/install/share/ihmc_gazebo
(untested command: `cp -r ~/catkin_ws/src/ihmc_repos/ihmc_gazebo/launch ~/catkin_ws/install/share/ihmc_gazebo/launch`)

Now run:

```
roslaunch ihmc_repos/ihmc_atlas_ros/launch/ihmc_atlas_gazebo.launch
```

Gazebo should start

Every time you catkin_make you will need to move that launch folder in order for catkin_make to work
The final command is now the command you must use to start gazebo

If you get an error including this: "Invalid <arg> tag: ihmc_gazebo", perform the install instructions again. Wget may have failed the first time

If gazebo launches but has a lot of squares in the environment and then crashes (or not), run:

```
source ~/catkin_ws/install/share/drcsim/setup.sh
```

If no robot shows up, you may need to catkin_make install

If gazebo is crashing, you can also just try running this:

```
sudo apt-get update && sudo apt-get install git \
  g++ vim nano wget  ca-certificates  ssh ros-kinetic-pcl-ros \
  x11vnc xvfb icewm lxpanel iperf xz-utils cmake screen terminator
konsole\
  ros-kinetic-pcl-conversions ros-kinetic-moveit \
  ros-kinetic-trac-ik ros-kinetic-footstep-planner \
  ros-kinetic-humanoid-localization ros-kinetic-multisense-ros \
  ros-kinetic-laser-assembler ros-kinetic-robot-self-filter \
  ros-kinetic-tf2-geometry-msgs ros-kinetic-joint-state-publisher \
  ros-kinetic-octomap-server ros-kinetic-octomap
ros-kinetic-gazebo-ros-control \
  ros-kinetic-joint-trajectory-controller ros-kinetic-controller-manager \
  ros-kinetic-image-transport \
  ros-kinetic-joint-state-controller ros-kinetic-position-controllers \
  ros-kinetic-sbpl \
  ros-kinetic-humanoid-nav-msgs ros-kinetic-map-server
ros-kinetic-trac-ik* \
  ros-kinetic-multisense-ros ros-kinetic-robot-self-filter
ros-kinetic-octomap \
  ros-kinetic-octomap-msgs ros-kinetic-octomap-ros ros-kinetic-gridmap-2d
\
  software-properties-common python-software-properties debconf-i18n \
  ros-kinetic-stereo-image-proc python-vcstool python-catkin-tools
ros-kinetic-ros-base \
  openjdk-8-jdk openjfx ros-kinetic-effort-controllers
ros-kinetic-camera-info-manager \
  ros-kinetic-gazebo-plugins ros-kinetic-fiducials
ros-kinetic-moveit-visual-tools
```

If you try to run gazebo and it crashes with the lots of errors related to PID and robot description, then it is probably because you are using a virtualized GPU. You should install your graphics drivers.

Appendix C: Every single thing that needs to be sourced

The order of sourcing is important! But I can't guarantee that this is the right order.

source ~/catkin_ws/install/share/drcsim/setup.sh

```
source /opt/ros/kinetic/setup.bash
source ~catkin_ws/install/setup.bash
source ~catkin_ws/devel/setup.bash
```

Appendix D: Common Commands

Run gazebo

```
roslaunch ihmc_atlas_ros ihmc_atlas_gazebo.launch use_local_build:=true
```

Run Footstep Planner:

Run Lidar:

The LiDAR must be started manually. First, the spindle speed of the LiDAR must be set using the command

```
rostopic pub /multisense/set_spindle_speed std_msgs/Float64 0.8
```

Next, the perception nodes must be launched using

```
roslaunch tough_perception_bringup field_laser_assembler.launch
```

To avoid excessive memory usage, the LiDAR will stop after publishing 5 maps. Each map takes a minimum of 8 seconds in real time to publish, which can be upwards of 30 seconds in simulation with a low real time factor. This command restarts the LiDAR:

```
rostopic pub /atlas/pause_pointcloud std_msgs/Bool false
```

Appendix E: Git Repository

https://github.com/ATLAS-MQP-20-21/ATLAS-MQP-2021.git

Appendix F: Project Presentation Day Video

https://www.youtube.com/watch?v=XpMTDYAgMRo